CISC 322/326 ASSIGNMENT 2: REPORT

GNUstep: Concrete Architecture Analysis

March 14, 2025

Group 18
Chi Ma (Presenter)
Nick He
Zhiming Jin
Tiantian Sang (Presenter)
Dunyi Xie
Lixing Yang (Leader)

Table of Contents

1 Abstract

This report provides an in-depth analysis of the derivation process and architecture of the GNUstep framework, focusing on its top-level structure and subsystem interactions. The derivation process involves key steps, including the identification of core components (libs-base, libs-gui, libs-back, CoreBase, and Developer Tools), dependency analysis, modular structuring, and concrete architectural verification using Understand. GNUstep follows a layered, modular architecture, where the GUI layer interacts with the Foundation layer and the Back subsystem to handle rendering and event processing.

A reflexion analysis comparing the conceptual and concrete architectures revealed unexpected dependencies, such as GUI relying on Developer Tools and CoreBase invoking GUI functions, which were not part of the original conceptual model. Additionally, the introduction of new components like Documentation and Distribution highlights the evolution of GNUstep's structure beyond its initial design. A subsystem analysis of NSView demonstrated deviations from the expected Model-View-Controller (MVC) paradigm, revealing a more integrated approach where rendering, event handling, and layout management are more tightly coupled. The 2nd-level subsystem analysis of NSView further explored its role in graphical rendering, user interactions, and its deeper integration with backend systems.

The report also presents two key use cases: designing an interface with Gorm and creating a new project in Project Center. These case studies illustrate real-world interactions between GNUstep components, demonstrating both the intended and unintended architectural dependencies. While the integration of additional dependencies increases complexity, it also enhances flexibility, performance, and development efficiency. This report ultimately highlights the trade-offs between theoretical design clarity and practical implementation requirements, providing insights into how GNUstep balances modularity, portability, and maintainability in its evolving architecture.

2 Introduction & Overview

2.1 Purpose of the Report

This report aims to analyze the top-level architecture of GNUstep, a free software implementation of the OpenStep standard. We investigate the conceptual and concrete architectures of the system, comparing expected theoretical structures with the actual implementation. Additionally, we examine the system's dependencies, module interactions, and unexpected component relationships. Through this study, we aim to understand the architectural strengths and challenges in GNUstep and provide recommendations for improving modularity and maintainability.

2.2 Organization of the Report

The report is structured as follows:

- Section 3: Derivation Process – Describes how we analyzed the architecture, including conceptual modeling, static analysis using Understand, and the challenges of mapping dependencies.
- Section 4: Top-Level Architecture – Presents the conceptual and concrete architectures of GNUstep, highlighting key components and their relationships.

- Section 5: Subsystem Analysis – Examines the libs-gui subsystem, detailing its structure, responsibilities, and dependencies.
- Section 6: Reflexion Analysis – Compares conceptual and concrete architectures, identifying unexpected dependencies and deviations from theoretical models.
- Section 7: Use Cases – Illustrates two real-world scenarios: designing an interface using Gorm and creating a new project in Project Center, showcasing component interactions in practice.
- Section 8: Lessons Learned – Summarizes key takeaways from our analysis and suggests improvements to GNUstep's architecture.

2.3 Key Findings

- Layered but Complex Architecture: While GNUstep follows a structured layered model, real-world implementation introduces bidirectional dependencies for efficiency, affecting modularity.
- Unexpected Dependencies: Our Understand-based static analysis identified direct dependencies between GUI and Developer Tools, CoreBase and GUI, and GUI and Distribution, which were not part of the original conceptual design.
- New Components: The Documentation and Distribution subsystems emerged as critical parts of the architecture, aiding developer guidance and software packaging, respectively.
- Use Case Analysis: The two use cases reveal how GNUstep components interact in real-world development scenarios, showing both intended and unintended architectural relationships.

By providing an in-depth look into GNUstep's actual software architecture, this report highlights areas where the system deviates from its intended modular design. These findings can help developers improve GNUstep's maintainability while optimizing performance.

3 Derivation Process

To begin our derivation process, we revisited the conceptual architecture design from our previous assignment. By reviewing GNUstep's official documentation, research papers, and developer discussions, we structured the conceptual architecture into its core components: libs-base, libs-gui, libs-back, CoreBase, and Developer Tools. We initially hypothesized a well-defined layered architecture with clear separations between these components. Based on this, we mapped the expected dependencies among them, ensuring that each module maintained a single responsibility within the system.

After loading the GNUstep source code into Understand, we generated concrete architectural views and compared them with our conceptual design. This analysis revealed several key differences, particularly in the interactions between modules. While the high-level structure aligned with our expectations, we found unexpected dependencies, such as direct calls between the GUI layer and Developer Tools. The derivation process became increasingly complex as we identified how different classes and functions interacted across subsystems. Mapping these

relationships required careful code inspection and an iterative refinement of our dependency diagrams.

The dependency analysis further highlighted challenges in fully encapsulating modular boundaries within GNUstep. Certain subsystems, like libs-gui and CoreBase, exhibited bidirectional dependencies that were not initially accounted for in our conceptual model. By refining our architecture diagrams and aligning them with actual code structures, we produced a clearer and more readable architectural representation. This approach not only deepened our understanding of the system but also allowed us to assess the real-world deviations from a purely layered architecture.

4 Top-level Architecture

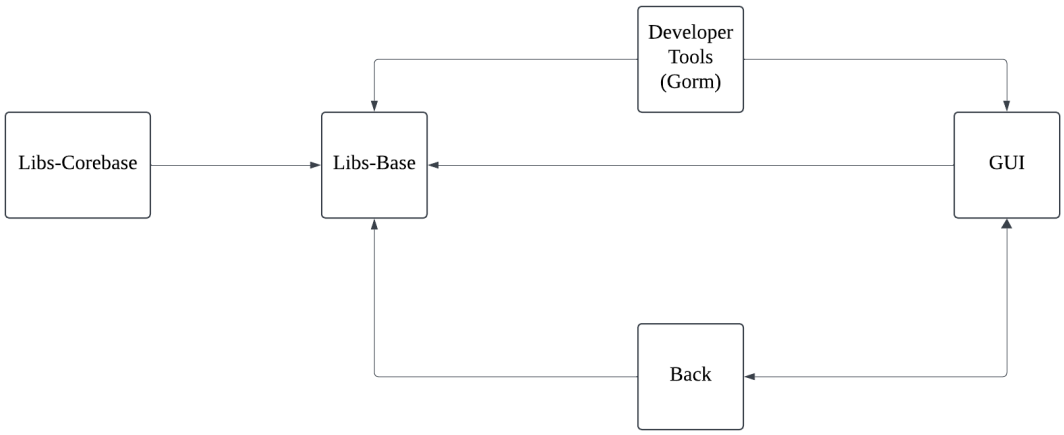

Figure 1: Recap of Conceptual Architecture

4.1 Conceptual Architecture

GNUstep's conceptual architecture is layered and modular, enabling cross-platform, OpenStep-like development in Objective-C. Foundation (GNUstep-base) handles core data structures, file I/O, and networking, forming the basis on which higher layers build. Above Foundation lies the GUI layer (GNUstep-gui), which defines windows, views, and controls, while delegating low-level rendering to the Back subsystem (GNUstep-back). This Back layer converts GUI commands into platform-specific calls (e.g., X11, Windows GDI), ensuring portability. Developer Tools (like GNUstep Make, Gorm, and ProjectCenter) streamline building and interface design, themselves leveraging Foundation and GUI capabilities.

Additionally, CoreBase was integrated after we learned about its potential from another great team's project. CoreBase extends GNUstep's scope with specialized functionalities, such as advanced data handling or cryptographic features, complementing Foundation without duplicating its fundamental services. Collectively, these components—Foundation, GUI, Back, Developer Tools, and CoreBase—embody a cohesive architecture where each part fulfills a focused role, resulting in a robust, flexible environment for building Objective-C applications across diverse platforms.

4.2 Concrete Architecture

As noted in our report on the system's conceptual architecture, GNUstep primarily follows a layered software architecture style. This hypothesis was confirmed by an inspection of the concrete architecture. The following dependency diagram is generated using Understand.

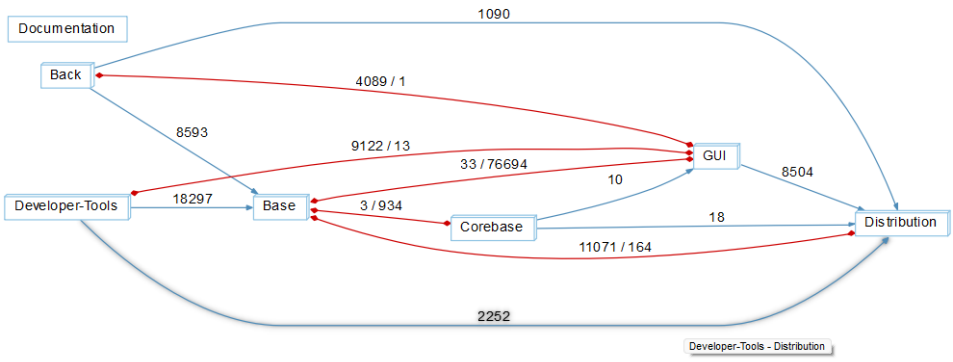

Figure 2: Dependency Diagram

The concrete architecture of GNUstep reflects its layered software architecture, as confirmed by an analysis of its dependencies. This structure ensures that each component interacts systematically while maintaining modularity, reusability, and platform independence.

At the foundation, Libs-Corebase and Libs-Base provide essential functionalities such as memory management, object handling, and basic system services. These libraries form the core layer that supports the upper layers. Directly above, Libs-GUI relies on these base libraries to implement GUI components like windows, buttons, and input handling.

The Libs-Back layer acts as an abstraction between the GUI and the system’s rendering backends. It ensures that GNUstep applications can operate across different platforms, such as X11, Windows, and macOS, without requiring changes to the GUI logic. This separation allows the same applications to run smoothly across multiple environments.

On the application level, tools like Gorm (a GUI designer) and Project Center (an integrated development environment) provide user-friendly interfaces for designing and managing applications. These tools interact with lower-level libraries but remain independent, reinforcing the modular nature of GNUstep’s architecture.

The dependency diagram, generated using Understand, confirms this structured hierarchy by showing how each component builds upon the foundational layers. This concrete architecture validates the layered design approach and ensures that GNUstep remains a flexible, portable, and scalable system.

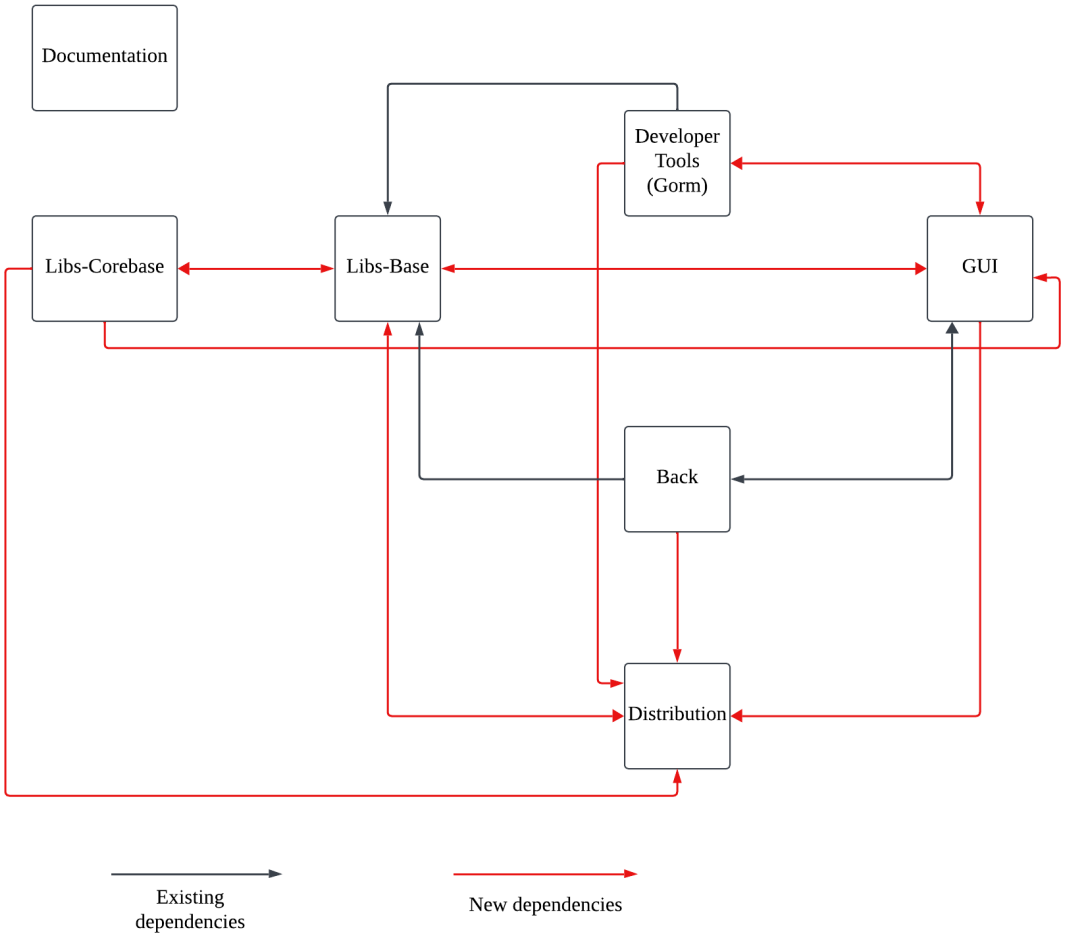

Figure 3: Concrete Architecture

4.3 Reflexion Analysis on the Top-level Architecture

Looking back at our team's initial conceptual architecture, we can see that the concrete architecture contains many unexpected dependencies and two new components.

4.3.1 New Components:

- Documentation centralizes all the guides, API references, and user manuals. It helps developers understand how to use Foundation APIs, how to design interfaces via Gorm, or how to set up build scripts in GNUstep Make.
- Distribution encompasses the process of packaging GNUstep frameworks and applications into runnable bundles or installers for end-users. It leverages the “common code” (within

Foundation) for file operations, versioning, or resource manipulation to ensure consistent deployment.

4.3.2 Unexpected Dependencies:

GUI → Developer Tools

The GUI layer relies on certain routines provided by Developer Tools (e.g., Gorm) for streamlined UI parsing and initialization, reusing logic rather than duplicating it. This approach simplifies the GUI's codebase but establishes a direct dependency on higher-level tools.

Base → GUI

The Base layer calls certain GUI routines—for instance, to handle UI-related preferences or configuration—thereby creating a direct dependency on the higher-level GUI code.

Corebase → GUI

CoreBase occasionally invokes GUI classes for specialized data handling processes that require visual feedback, thereby linking CoreBase to the GUI layer.

GUI → Distribution

The GUI layer calls on Distribution to bundle UI resources into final builds, ensuring graphics and interface elements are included in released packages. This approach streamlines resource management but also ties GUI functionality to the packaging process.

Developer → Distribution

The Developer Tools rely on Distribution to package freshly built binaries and resources, ensuring streamlined release processes across platforms. This integration ties the build environment directly to final deployment.

Base ↔ Distribution

The Base library collaborates with Distribution for resource packaging, leveraging file and metadata capabilities at build time. In turn, Distribution relies on Base's abstractions to handle core operations consistently across platforms.

Corebase → Distribution

CoreBase uses Distribution to package its specialized data processing or cryptographic components into final builds, integrating these advanced features seamlessly. This arrangement ensures that CoreBase functionalities are included and managed consistently across release processes.

Corebase ↔ Base

CoreBase depends on Base for fundamental classes, data structures, and concurrency features, allowing it to focus on more specialized functionalities. Conversely, Base can tap into CoreBase's advanced routines (e.g., cryptographic services), thereby leveraging its specialized capabilities. This mutual reliance ensures that each library can benefit from the other's strengths, creating a richer overall framework.

5 Subsystem Analysis: libs-gui

Conceptual View：

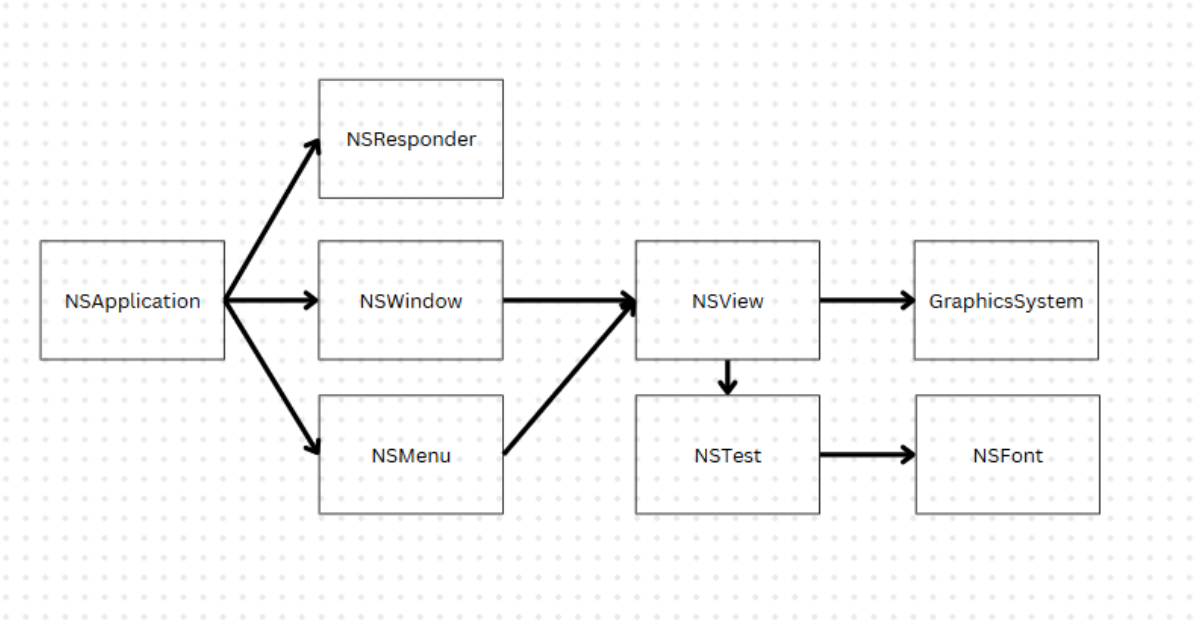

Figure 4: Conceptual Architecture

Concrete View：

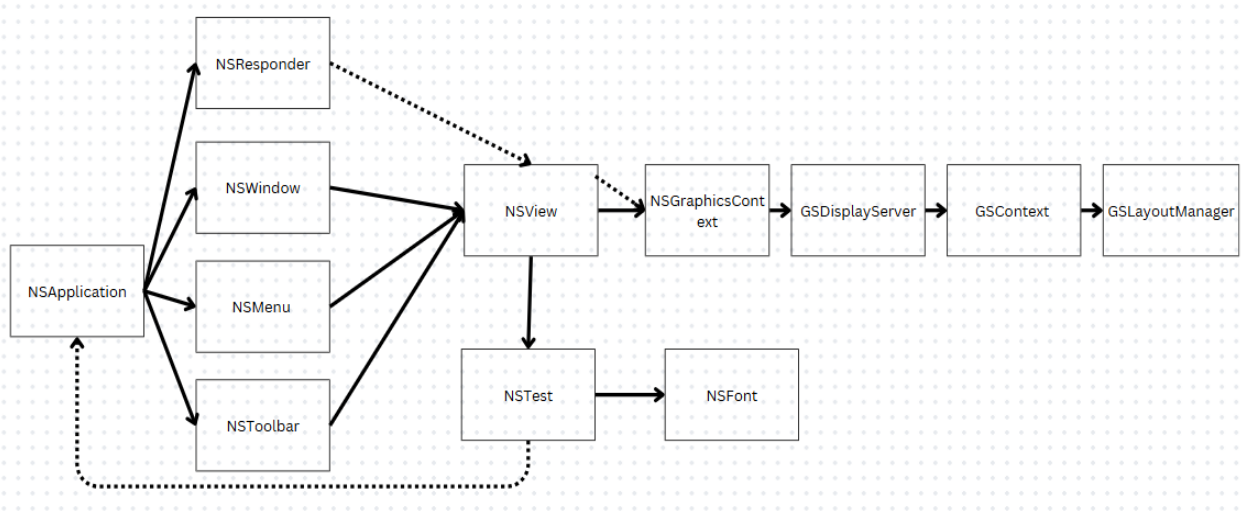

Figure 5: Concrete Architecture

5.1 Internal Structure and Interactions：

NSApplication:
Main Responsibilities: Manages the application's lifecycle, including event loops, user input delegation, and coordination between UI elements. It ensures that events are processed efficiently and updates are reflected in the graphical interface.
Dependencies: Communicates with NSResponder for event handling, manages multiple NSWindow instances, and updates UI controls through NSMenu and NSToolbar. It also interacts with NSText in the concrete architecture, introducing an unexpected dependency on text management.

NSResponder:
Main Responsibilities: Handles and delegates user input events, including keyboard, mouse, and gesture interactions. It was expected to be the central handler for all events in the conceptual model.
Dependencies: Receives raw input from the system, forwarding events to NSApplication, NSWindow, or NSView. In the concrete implementation, certain UI components override centralized event handling, directly processing input without passing through NSResponder.

NSWindow:
Main Responsibilities: Manages the application's windows, controlling their size, visibility, and user interactions. It acts as a container for multiple NSView components.
Dependencies: Contains and manages NSView instances, updates its state through NSApplication, and processes user interactions relayed from NSResponder.

NSView:
Main Responsibilities: Handles UI rendering and layout updates. It manages resizing, clipping, and layering of interface components.
Dependencies: Uses NSGraphicsContext for rendering in the concrete model instead of directly communicating with a graphics system. It interacts with NSText for displaying textual content and receives updates from NSMenu and NSToolbar.

NSGraphicsContext:
Main Responsibilities: Acts as an abstraction layer between NSView and the lower-level graphics system. It manages drawing operations, coordinate transformations, and rendering optimizations.
Dependencies: Works with GSDisplayServer to send rendering instructions and ensure efficient drawing. It was introduced in the concrete implementation, replacing the direct NSView → Graphics System connection assumed in the conceptual model.

GSDisplayServer：
Main Responsibilities: Handles communication between the UI system and the underlying graphics layer. It processes rendering requests from NSGraphicsContext and manages drawing commands.
Dependencies: Interacts with GSContext to maintain rendering state and optimize display performance.

GSContext:
Main Responsibilities: Provides an abstraction for managing graphical output, ensuring compatibility between the UI components and the graphics backend.
Dependencies: Receives instructions from GSDisplayServer and works with GSLayoutManager to adjust UI layouts.

GSLayoutManager:
Main Responsibilities: Manages UI layout calculations, including positioning, alignment, and responsiveness of visual elements.
Dependencies: Receives layout information from GSContext and ensures that all UI components are properly arranged on the screen.

NSText:
Main Responsibilities: Handles text rendering and layout, supporting multi-line formatting, rich text styles, and font management.
Dependencies: Uses NSFont for typography but also communicates with NSApplication in the concrete architecture, introducing a dependency that was not present in the conceptual model.

NSFont:
Main Responsibilities: Provides font resources, including different typefaces, sizes, and weights, ensuring consistent text rendering.
Dependencies: Used by NSText to format and render textual content.

NSMenu & NSToolbar:
Main Responsibilities: Manages user-accessible controls such as menus and toolbars, allowing interaction with the application through structured UI components.
Dependencies: Interacts with NSApplication for state management and dynamically updates UI elements via NSView.

5.2 Reflection Analysis:
The Figure 5 shows the concrete implementation of the GNUstep GUI subsystem, the key differences are as follows:

- More Detailed Subdivisions: the conceptual model assumed NSView would directly communicate with a Graphics System, but additional components were introduced.
    - NSGraphicsContext was added as an intermediary between NSView and GSDisplayServer.
    - GSDisplayServer, GSContext, and GSLayoutManager were introduced to handle rendering in multiple stages.
- Additional Dependencies: The conceptual model assumed centralized event handling and direct rendering, but the actual implementation introduced more connections for better performance.
- Bidirectional Dependencies: Some modules interact in both directions, increasing coupling but improving efficiency.

- NSView and NSGraphicsContext exchange rendering state information.
- GUI components reference the Graphics System multiple times, making rendering more dynamic but complex.

The conceptual model focused on a clear separation of concerns, but the real implementation introduced extra components and dependencies for better modularity, flexibility, and performance. While this adds complexity, it also allows for a more optimized and responsive UI.

6 2nd-level Subsystem Analysis: NSView Subsystem

The NSView subsystem is a key part of the GNUstep GUI framework. It is responsible for managing the visual content of applications, including layout, rendering, and user interaction for all visible elements in a window. It sits between NSWindow (which manages the container) and the rendering back-end (which actually draws things to the screen), and works together with event-handling classes like NSResponder. In our team, we chose to analyze this subsystem in depth because it connects both logic and visual components, and plays a central role in the user interface.

The NSView subsystem belongs to the GUI layer of GNUstep's architecture, layered above the Foundation library and interacting closely with the rendering layer handled by the Back library. Our group originally assumed that this system would follow a clean object-oriented architecture, applying MVC patterns and strict layer separations, especially between GUI elements and rendering logic. However, through code analysis using the Understand tool and reviewing libs-gui source code, we found that the real implementation is more integrated and complex than expected. Below, we present our findings through conceptual architecture, concrete structure, and a reflexion analysis that compares the two.

6.1 Conceptual Architecture

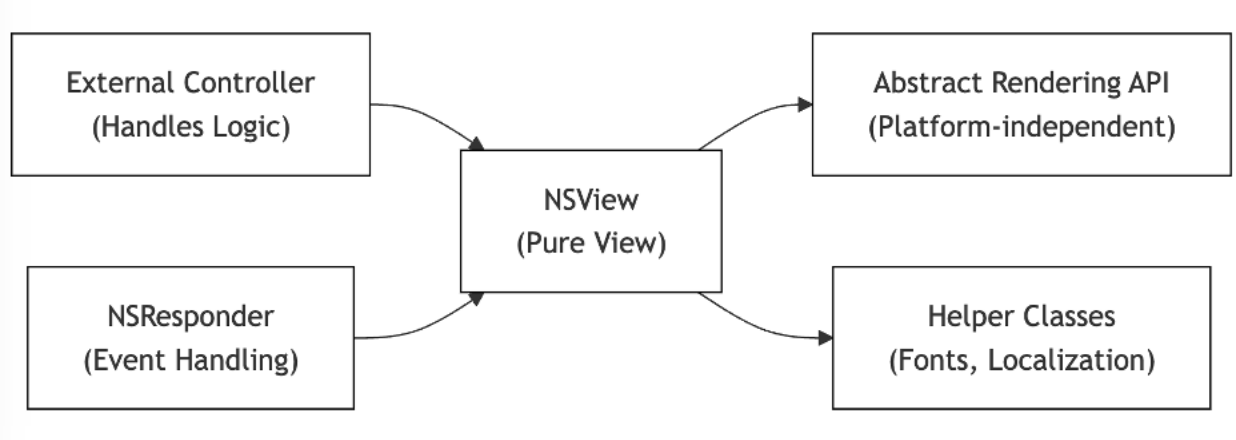

Figure 6: Conceptual Architecture

In our A1 report, we outlined a conceptual view of the GUI subsystem based on modular design and separation of concerns. In that model, NSView was designed as a pure "View" in the Model-View-Controller architecture. It should only be responsible for presenting visual content, with no logic of control or data management. NSView would receive input from a controller class and display information from model objects.

We also expected NSView to work with an abstract rendering system through standardized APIs. Rendering tasks would be passed down to a separate platform-dependent rendering system via interfaces, keeping NSView fully platform-independent. Input events would be received by NSResponder, which would then pass them to the appropriate view or controller. Resources like fonts and localized strings would be accessed through the Foundation layer or via helper classes, never by directly reaching into lower-level APIs.
This conceptual architecture aimed to preserve portability, modularity, and maintainability by clearly separating logic, view, and rendering.

6.2 Concrete Architecture

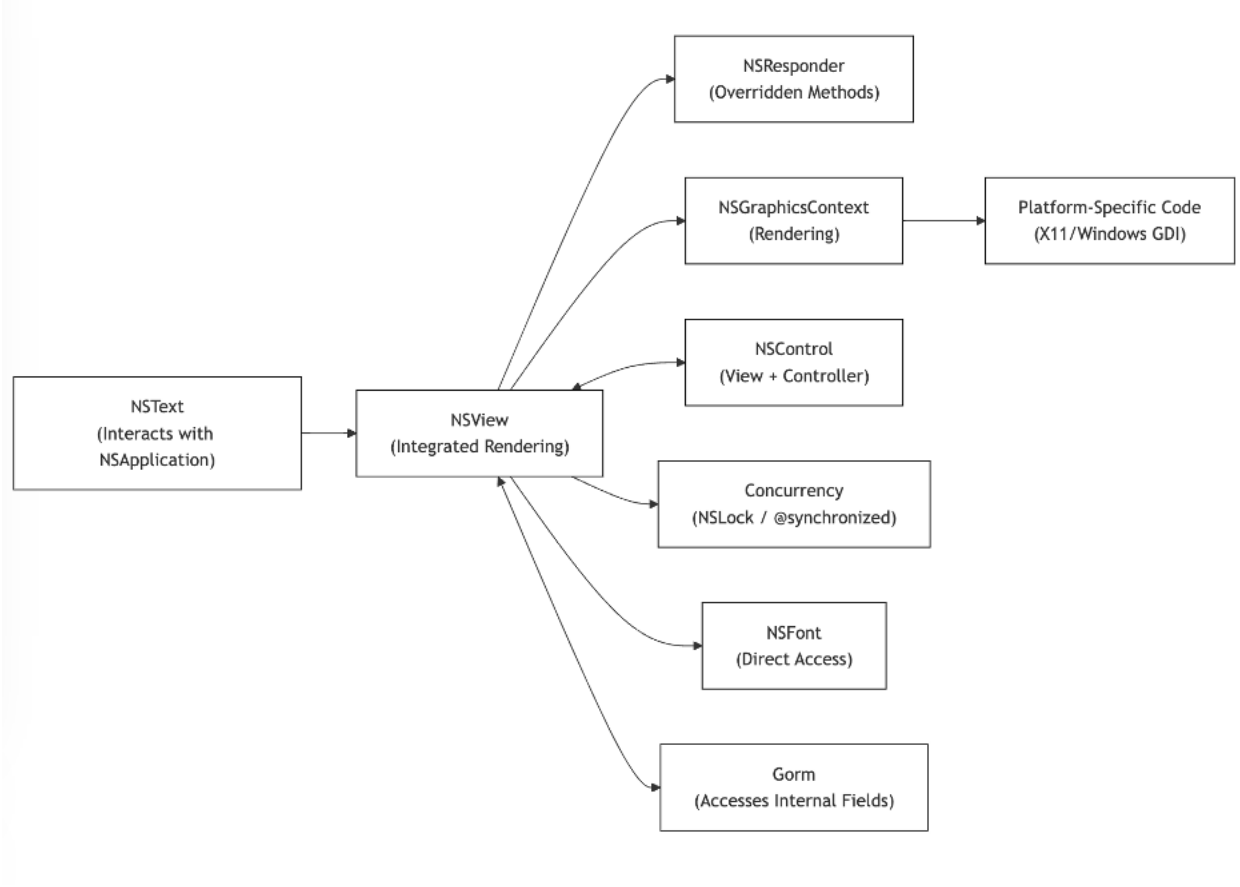

Figure 7: Concrete Architecture

Diagram Explanation

NSView (Integrated Rendering):Handles its own rendering by calling NSGraphicsContext (no strict abstraction layer).

NSGraphicsContext → Platform-Specific Code:Communicates with OS-level APIs (X11, Windows GDI), tying NSView more tightly to the platform.

NSControl (View + Controller):Combines view and control logic, creating a bidirectional relationship with NSView.

Concurrency:NSView (and related classes) may spawn threads or use locking to manage background tasks.

NSFont (Direct Access):NSView directly pulls font information, rather than delegating this to helper layers.

NSText → NSApplication:Text components sometimes interact at the application level, an unexpected extra dependency.

Gorm:Accesses internal fields of NSView (bypassing public APIs), indicating tighter coupling.

NSResponder (Overridden Methods):
NSView subclasses override event handling, mixing view and responder responsibilities.

In the actual implementation of libs-gui, the NSView subsystem takes on more responsibilities than expected. It handles layout, updates, and even performs its own rendering in many cases. Instead of working with an abstract graphics layer, NSView interacts directly with NSGraphicsContext, which is a class responsible for translating drawing commands into backend-specific rendering (like X11 or Windows GDI). This means that NSView is not fully isolated from the platform as originally expected.
Additionally, NSView subclasses often override event-handling methods from NSResponder, making them both view and controller in practice. For example, NSControl, a subclass of NSView, includes logic for handling user interactions and responding to them internally. This breaks the expected MVC separation.
We also found that some NSView implementations include conditional macros like #ifdef to deal with different backends. This introduces platform-specific code into a part of the system that was meant to be abstract and platform-independent.
Finally, concurrency management is not centralized in NSRunLoop as we assumed. Some operations in NSView or its related classes spawn their own threads and use synchronization mechanisms like NSLock or @synchronized, which adds complexity and risks such as race conditions or deadlocks.

6.3 Reflexion Analysis
Missing Dependencies:
NSView → NSGraphicsContext: In the conceptual model, we imagined an abstract rendering API. In reality, NSView communicates directly with NSGraphicsContext.
NSView → NSFont: Text rendering within views directly accesses font data, which wasn't shown in the conceptual model.
NSText → NSApplication: We didn't anticipate that text rendering components would need to interact with the top-level application class.

Unexpected Dependencies:
NSView → Platform-specific code: Platform-specific drawing is embedded directly into NSView, which breaks the clean layering expected.
NSControl → Controller Logic: Instead of separating control logic, some subclasses of NSView like NSControl handle input and updates themselves.
Gorm → Internal Fields of NSView: Gorm uses private fields of NSView classes, indicating tighter coupling than expected.

Removed Dependencies:

NSView → Abstract Graphics Layer: In conceptual design, we had an abstract graphics interface layer. This does not exist in the concrete implementation.
NSView → External Controller: The controller is sometimes bypassed entirely in the actual code, especially in interactive views.

7 Use Case

Use Case 1: Creating a new project in Project Center

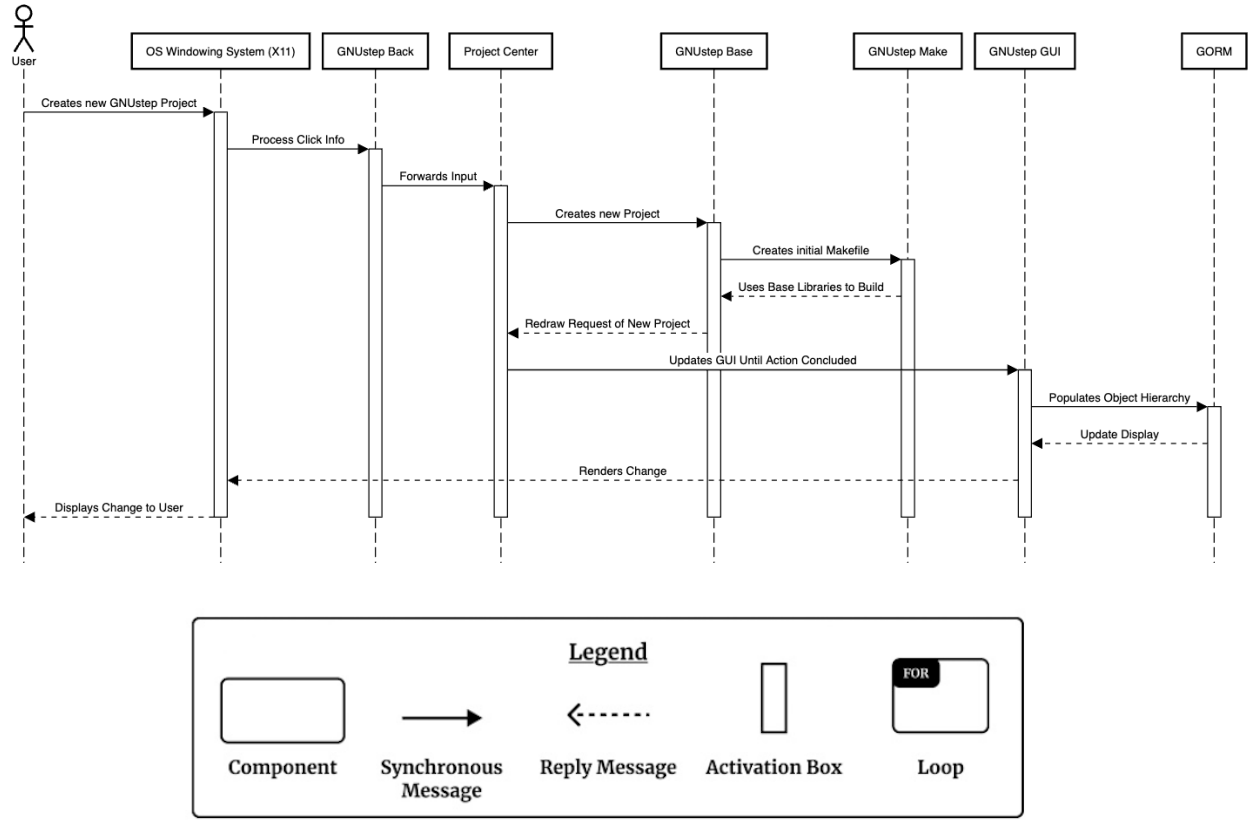

Figure 8: UML Sequence Diagram of Use Case 1

The sequence diagram illustrates the process of creating a new project within GNUstep's Project Center. This interaction involves multiple components, including the OS Windowing System (X11), GNUstep Back, Project Center, GNUstep Base, GNUstep Make, GNUstep GUI, and GORM. The flow captures the user-initiated project creation, system-level processing, GUI updates, and rendering of changes.

The process begins when the User initiates a new GNUstep project. This request is received by the OS Windowing System (X11), which processes the user's click event. The GNUstep Back component then forwards this input to the Project Center, where the actual project creation process begins.

Once the Project Center receives the request, it sends a command to GNUstep Base to create a new project. The GNUstep Base then delegates the build process to GNUstep Make, which creates an initial Makefile and integrates the Base Libraries needed for project compilation. Once the necessary libraries are loaded, the system sends a redraw request back to the Project Center, ensuring that the new project structure is reflected in the GUI.

After the project is created, the GNUstep GUI component is responsible for updating the interface dynamically. It continues to update until the process is complete. At this point, GORM, the interface designer, is notified to populate the object hierarchy within the GUI. This ensures that all UI elements of the newly created project are properly structured.

Finally, the GUI sends an update display request, which propagates back through the system, leading to the rendering of the new project interface. The OS Windowing System (X11) ensures that the user sees the rendered changes, and the process concludes when the User receives the final project layout.

Use Case 2: Designing an Interface with Gorm

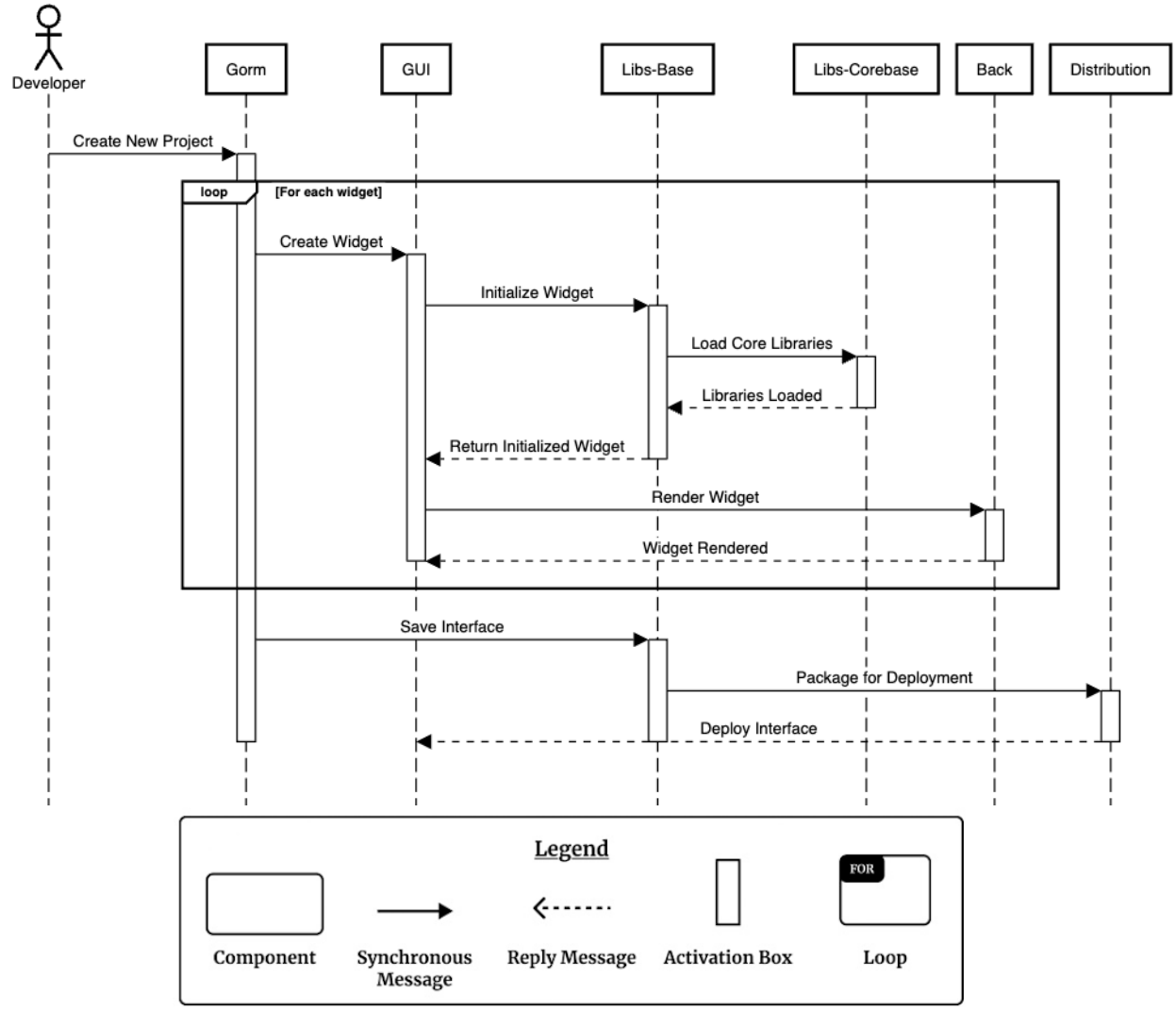

Figure 9: UML Sequence Diagram of Use Case 2

The sequence diagram illustrates the process of designing a graphical user interface (GUI) using Gorm in GNUstep. The Developer starts by creating a new project in Gorm, which triggers the system to enter a loop for adding multiple widgets.

For each widget, Gorm requests GUI to create it, which then sends an initialization request to Libs-Base. If necessary, Libs-Base loads Core Libraries from Libs-Corebase. Once the widget is initialized, it is returned to the GUI for rendering, and Back handles the rendering process.

After completing the UI design, Gorm saves the interface, which is then packaged for deployment by Libs-Base and Distribution. Finally, the GUI deploys the interface, making it available for use.

8 Lesson Learned

The conceptual model provided a straightforward structure with clear, direct dependencies. However, the concrete implementation introduced multiple abstraction layers to enhance modularity and performance. This demonstrates that while conceptual designs prioritize clarity and separation of concerns, real-world implementations often require additional complexity to address practical challenges such as rendering efficiency and event handling flexibility.
Through our analysis of the NSView subsystem, we observed significant divergence between idealized architectural models and practical implementations. While our A1 report emphasized strict modularity and adherence to the MVC pattern, our deeper investigation revealed that performance, usability, and code reuse often necessitate more integrated solutions. For example, NSView not only handles layout and rendering but also incorporates control logic and interacts directly with backend rendering systems like NSGraphicsContext. This tight coupling increases code complexity but enhances responsiveness and platform-specific optimization.
Additionally, we discovered that event handling and concurrency management in GUI systems are not always centralized or abstracted. Instead, direct thread operations and synchronization mechanisms are sometimes implemented at the view layer. Another key finding was that internal components like Gorm rely on private data fields, deviating from the expected clean, interface-based design.
In summary, GNUstep's layered architecture—spanning Base, GUI, Back, Developer Tools, CoreBase, and Distribution—balances modularity with necessary interconnections. While certain subsystems overlap unexpectedly, the clearly defined roles and well-documented APIs help maintain clarity and ease of maintenance. By integrating foundational utilities with specialized features, GNUstep remains both adaptable and robust for Objective-C development across diverse platforms.
Ultimately, this analysis highlights a critical lesson: conceptual models are invaluable for architectural planning, but real-world implementations must balance theoretical ideals with practical constraints like performance, platform compatibility, and developer convenience.

9 Reference

GNUstep. GUI Library Reference Manual. Retrieved from https://www.gnustep.org/resources/documentation/Developer/Gui/Reference/

GNUstep. GNUstep Developer Documentation. Retrieved from https://www.gnustep.org/developers/documentation.html

GNUstep. GNUstep-gui Source Code Repository. Retrieved from https://github.com/gnustep/libs-gui